Characterizing Kernel Malware Behavior with Kernel Data Access Patterns

Junghwan Rhee, Zhiqiang Lin, Dongyan Xu
Department of Computer Science and CERIAS, Purdue University
West Lafayette, IN 47907
{rhee,zlin,dxu}@cs.purdue.edu

ABSTRACT

Characterizing malware behavior using its control flow faces several challenges, such as obfuscations in static analysis and the behavior variations in dynamic analysis. This paper introduces a new approach to characterizing kernel malware's behavior by using kernel data access patterns unique to the malware. The approach neither uses malware's control flow consisting of temporal ordering of malware code execution, nor the code-specific information about the malware. Thus, the malware signature based on such data access patterns is resilient in matching malware variants.

To evaluate the effectiveness of this approach, we first generated the signatures of three classic rootkits using their data access patterns, and then matched them with a group of kernel execution instances which are benign or compromised by 16 kernel rootkits. The malware signatures did not trigger any false positives in benign kernel runs; however, kernel runs compromised by 16 rootkits were detected due to the data access patterns shared with the compared signature(s). We further observed similar data access patterns in the signatures of the tested rootkits and exposed popular rootkit attack operations by ranking common data behavior across rootkits. Our experiments show that our approach is effective not only to detect the malware whose signature is available, but also to determine its variants which share kernel data access patterns.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Kernel Malware Analysis, Kernel Malware Signature, Kernel Data Access Patterns

1. INTRODUCTION

Characterizing malware behavior is a non-trivial research problem and there have been many approaches to address its challenges. A large body of work uses malware's control flow patterns, such as instruction sequences or system-call sequences, to detect or analyze malware [2, 3, 9, 14, 15]. In response to such approaches, malware often employs various obfuscation techniques to confuse malware analyzers [8, 10, 25, 26]. Meanwhile, these approaches face challenges arising from execution dynamics, such as dynamic code paths and the impact of other system components (e.g., network latency and signals), which can cause variations in the characterized malware patterns. The situation is more complicated in the kernel space because operating system (OS) kernels have a highly dynamic workload, including interrupts, the coordination of user processes, and the management of low level resources (e.g., page tables).

For detection and prevention of kernel malware, there is another collection of work called the code integrity-based approach [23, 24]. This approach allows only authorized code for execution and considers any code outside the white list as malicious. Therefore, this approach is effective for kernel rootkits that introduce new code to kernel space. However, other advanced rootkits perform the attacks by exploiting only legitimate kernel code (e.g., the usage of memory devices [20], kernel bugs, and return-oriented programming [13]); and such attacks are not properly handled by this approach. In addition, this approach authorizes kernel driver code based on policies trusting OS developers or venders without systematic examination of the code. For example, existing code integrity-based approaches [24, 23] allow the kernel text and a list of benign kernel modules included in the OS distributions. These policies do not provide safety from hidden malicious code inside the authorized code. Thus the capability of examining kernel drivers for potentially malicious behavior regardless of such policies is desirable.

In this paper, we introduce an alternative approach that characterizes kernel malware behavior by using its data access patterns. The idea is that when kernel malware tampers with core kernel data, there exist unique kernel data access patterns. As such, we could take a subset of data access patterns that consistently appears in multiple kernel execution instances only when the malware is active and generate the malware signature using the subset.[1] These patterns under constraints neither include malware's temporal control

[1]We use the terms "a kernel execution instance" and "a kernel run", to represent an instance of the OS kernel execution, which starts from its booting and ends at its shutdown.

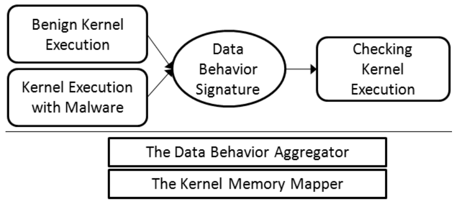
Benign Kernel Execution
Kernel Execution with Malware
Data Behavior Signature
Checking Kernel Execution
The Data Behavior Aggregator
The Kernel Memory Mapper

Figure 1: Overview of DataGene.

flow information, nor the code-specific information about the malware. Therefore, this approach is less susceptible to obfuscations and more effective for matching malware variants.

To evaluate the effectiveness of our approach, we generated the signatures of three classic rootkits and matched them with benign kernel runs and malicious kernel runs where the rootkits are active. This experiment did not trigger any false positives in benign runs, but it did detect the presences of the 16 kernel rootkits which have a variety of attack goals and mechanisms. We further analyzed the data behavior of such rootkits and found that a majority of them exhibit shared behaviors to one another. We argue that such common behavior can be used to effectively detect malware variants (e.g., polymorphic rootkits, different versions, and similar rootkits). Also, if an unknown malware shares a data operation with any data behavior of existing rootkits, its presence can be determined.

The contributions of this paper are as follows:

- We present a complementary approach that characterizes kernel malware behavior by using its unique data access patterns. This approach can be applied to detect kernel rootkits that do not violate kernel code integrity.

- This approach can automatically construct malware signatures by using a binary-only malware program. Malware behavior is extracted by capturing a subset of kernel behavior that consistently appears across kernel execution instances only when the malware is active.

- This signature uses data behavior with generalized code information and does not involve control flow of malware code execution. Hence it can detect the variants of kernel malware by exposing similar data behavior across kernel malware.

We have implemented a prototype called DataGene based on our approach. DataGene is mainly designed for non-production systems such as a honeypot for kernel malware and a malware analysis system. For instance, when a new proprietary driver is deployed, DataGene can inspect it for potential hidden malicious behavior similar to the behavior observed in existing kernel malware. Also for a newly distributed kernel malware sample, if it shares any data behavior with existing kernel malware, DataGene can detect it and extract its data behavior, which can be used to detect this malware and its variants. In addition, DataGene can detect challenging kernel rootkits that do not violate kernel code integrity. Therefore, this data-oriented approach can complement the code integrity-based approach in the defense against kernel malware.

2. DESIGN OF DATAGENE

In this section, we present the design of DataGene that characterizes the behavior of kernel malware and determines its presence based on data access patterns. As DataGene uses information regarding memory accesses, for convenience our design employs virtual machine techniques to capture the accesses. The overview of DataGene is presented in Figure 1, and the components of this system are as follows.

As a basic unit to represent the kernel's data behavior, DataGene generates a summary of the access patterns for all kernel objects accessed in a kernel execution instance. To identify dynamic kernel memory objects, this process takes advantage of a kernel memory mapping process (shown as The Kernel Memory Mapper in Figure 1). For each access on kernel memory in the guest OS, the virtual machine monitor (VMM) intercedes and records the information of the kernel memory access, such as the accessing code, the accessed memory type, and the accessed offset (The Data Behavior Aggregator).

To determine the malware behavior, the memory access patterns for two kinds of kernel execution instances are generated: benign kernel runs and malicious kernel runs where kernel malware is active. By taking the difference between the two sets of memory access patterns, we estimate the unique data behavior incurred from the kernel malware and generate its signature (Data Behavior Signature). In order to detect a kernel malware, the generated signature is compared to the memory access patterns of a tested kernel execution instance (Checking Kernel Execution).

2.1 Data Behavior Profile Approach

In this section, we present basic terminologies that represent the memory access patterns of kernel execution. A *data behavior profile* (D_r) is a set of memory access patterns for kernel data structures accessed in a kernel execution instance r. An element of this set is called a *data behavior element* or simply an element (e), and it is defined as a quintuple (5-tuple): the address of the code that accesses memory (c), the kind (read or write) of memory access (o), the kind (static or dynamic) of the accessed memory (m), the class of the accessed memory (i), and the accessed offset(s) (f) inside the memory of the class i.

$$e = (c, o, m, i, f)\,, D_r = \{e | e \text{ for static and dynamic kernel objects}\}$$

c is the address of the kernel code that reads or writes kernel memory. o represents the kind of memory access which is 0 for a memory read and 1 for a memory write.

The kind of the accessed memory, m, is 0 for a dynamic object and 1 for a static object. The class i is defined differently, depending on the memory kind. Static objects are known at compile time; therefore, we are able to assign unique numbers as their identifiers. A class of a static object can represent either a static data object or a kernel function in the kernel text. In the case of dynamic kernel objects, there are multiple memory instances for the same data type at runtime. Dynamic kernel objects allocated by the same code correspond to the data instances of the specific data type used in the allocation code. Thus, we aggregate the access patterns of dynamic kernel objects that share the allocation code. The address of this code (called an allocation code site) is used as a unique class for such objects.

f is an offset, or a range of offsets, accessed by the code at c. We allow a range of offsets because if this object is an

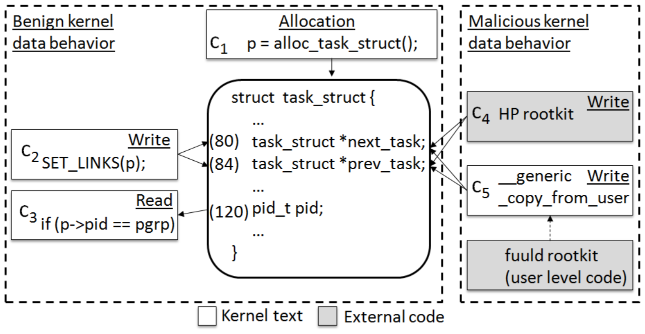
Benign kernel data behavior
Allocation
C1 p = alloc_task_struct();
Malicious kernel data behavior
struct task_struct {
...
(80) task_struct *next_task;
(84) task_struct *prev_task;
...
(120) pid_t pid;
...
}
Write
C2 SET_LINKS(p);
Read
C3 if (p->pid == pgrp)
Write
C4 HP rootkit
C5 __generic _copy_from_user Write
fuuld rootkit (user level code)
Kernel text
External code

Figure 2: An example of kernel code in benign and malicious kernel runs.

array, the accessed offsets can vary for the same accessing code. Handling them as separate data behavior elements can cause a high number of elements with slightly different offsets for the same accessing code. To avoid this problem, we use a threshold to convert a list of elements whose offsets are different (but with the same accessing code) to an element with an offset range.

Figure 2 presents kernel code showing the examples of data behavior elements. The rounded box in the middle of Figure 2 shows a dynamic kernel object allocated by the code at the address c_1. This figure shows how this object is accessed by several code sites in kernel execution. Two fields, `next_task` (offset 80) and `prev_task` (offset 84), are written by the code at c_2. The code at c_3 reads the `pid` field (offset 120). Therefore, the data behavior elements for this code example are as follows.

$$(c_2, 1, 0, c_1, 80), (c_2, 1, 0, c_1, 84), (c_3, 0, 0, c_1, 120)$$

These elements are the access patterns in a benign kernel run. If kernel malware is active in this kernel, the access patterns can be extended due to the malware behavior. For instance, if kernel rootkits `hp` and `fuuld` are active as shown in the right-hand section of Figure 2, there would be additional accesses to the `next_task` and the `prev_task` fields by the code at c_4 and c_5. Consequently, the data behavior profile is extended with the additional elements as follows.

$$(c_4, 1, 0, c_1, 80), (c_4, 1, 0, c_1, 84), (c_5, 1, 0, c_1, 80), (c_5, 1, 0, c_1, 84)$$

Here c_4 represents the code of the `hp` rootkit, which is in the form of a kernel driver. The code integrity-based rootkit defense approach [23, 24] can determine this access as malicious based on the fact that this driver code is not in the authorized code list. In contrast, the code at c_5 is part of legitimate kernel code which is indirectly exploited to overwrite this data structure. This rootkit case does not violate kernel code integrity; therefore, the approach based on code integrity cannot detect this attack behavior.

In both cases, malware behavior uniquely appears only when the malware runs. Our approach aims to capture such unique behavior to determine the presence of malware.

2.2 Generating a Data Behavior Profile

In this section, we present the process for generating a data behavior profile, which summarizes the access patterns for all kernel objects accessed in a kernel run. Based on this information, we generate the signature of malware and inspect a kernel run for malicious data access patterns. A data behavior profile is generated based on two underlying functions. First, kernel objects should be identified with their unique classes. Second, the access patterns on numerous (e.g., tens of thousands in modern OSes) dynamic data instances should be summarized regarding their classes. We present two system components to provide these functions.

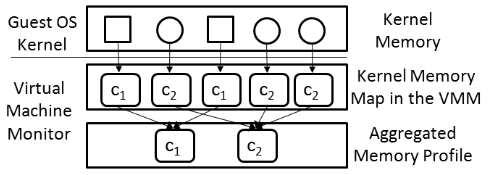

Figure 3: Aggregating memory accesses on dynamic kernel objects regarding their classes (allocation sites) c_1 and c_2.

The Kernel Memory Mapper. DataGene uses the patterns of memory accesses on kernel objects and requires a kernel memory mapping mechanism [1, 7, 19, 21, 22, 27] to identify the targets of kernel memory accesses. Among such approaches, LiveDM [21] provides runtime kernel memory mapping which enables the identification of a memory access' target. LiveDM identifies kernel objects by transparently capturing the allocation and deallocation events of kernel memory. The generated map maintains the allocation code for each dynamic object as its runtime identifier. In offline static analysis, this identifier can be automatically translated into a data type by traversing kernel source code. We implemented the *kernel memory mapper* by employing LiveDM's approach.

The Data Behavior Aggregator. In a kernel execution instance, there exist a varying number of dynamic kernel data instances. To compare the access patterns of dynamic kernel objects in different kernel runs, it is necessary to aggregate the memory accesses on such objects regarding their classes. The allocation code represents the instantiation of a data type at a specific code position. By using a memory allocation code site as the classifier of dynamic kernel objects, we can aggregate the access patterns of dynamic instances of the same type *and* of a similar usage.

Figure 3 illustrates this aggregation process. When a dynamic kernel object is allocated in a guest OS kernel, the kernel memory mapper stores its address range and the allocation code site as the class information in the kernel memory map. We have a memory mapping layer to aggregate the memory accesses on dynamic kernel objects regarding their data classes. Whenever kernel code reads or writes any dynamic kernel object, the VMM intercedes and identifies the targeted object by using its class information from the kernel memory map. If this memory access pattern is new, it is recorded in the aggregated memory profile.

2.3 Characterizing Malware Data Behavior

In this section we demonstrate how we characterize the behavior of kernel malware based on data behavior profiles. We first present the challenges and describe how we address them. Then, we describe the generalization process in the malware behavior to match similar behavior across different kernel malware. Next, we present an algorithm to match a malware signature with the data behavior profile of a kernel execution instance.

Challenges and Our Solutions. DataGene characterizes malware behavior by using the memory access patterns uniquely observed in malware execution. To estimate such information without requiring specific knowledge of malware, DataGene compares two kinds of kernel execution instances: benign kernel runs and malicious kernel runs with malware. This approach faces several challenges:

- **Variations in the Runtime Kernel Behavior.** Generally, the difficulty in obtaining a complete set of kernel execution paths is a well-known challenge for an approach based on dynamic execution. If we focus on the data behavior in benign execution, it is in fact a problem because the runtime kernel behavior is highly dynamic across different runs. However, we focus on the data behavior of kernel malware that consistently appears only when the malware is active. This behavior is a closed set of malware activity, and we use multiple instances of malicious kernel execution to capture the subset of malicious behavior that meets such constraints.

- **Irregular Access Patterns on Kernel Stacks.** Kernel stacks are kernel objects that have irregular access patterns. Whenever a kernel function is called or returns, the stack is accessed for various purposes such as return values, function arguments, and local variables. Since the kernel control flow is highly dynamic, the set of code sites that access the stack and the accessed offsets within the stack vary significantly. Also, the contents of kernel stacks are irregular at different runs. As such, a simple way to handle this problem is to exclude stacks from our analysis. The kernel memory mapper provides the identifier for kernel stacks and we solve this problem by removing the information for such dynamic objects from the analysis.

- **Varying Offsets in Arrays.** Some data structures (e.g., arrays and buffers) have a range of space, a part of which can be used at runtime. For example, the accessed offsets of a buffer can be different depending on the data contained in it. This problem is handled by using multiple instances of kernel execution. If the accessed offset of memory is different in each execution, it is not used for a malware signature because it may not be used in another run. Only the data behavior that occurs in a consistent pattern when malware is active becomes the candidate for the signature.

Characterizing Malicious Data Behavior. In order to reliably characterize the data behavior of kernel malware in dynamic execution, we use multiple kernel runs. $D_{M,j}$ is a data behavior profile for a malicious kernel run j with malware M. $D_{B,k}$ represents a data behavior profile for a benign kernel execution k. We apply the set operations on n malicious kernel runs and m benign runs as follows. The generated signature is called a *data behavior signature* for the malware M and shown as S_M.

$$S_M = \bigcap_{j \in [1,n]} D_{M,j} - \bigcup_{k \in [1,m]} D_{B,k}$$

This formula represents that S_M is the set of data behavior that *consistently* appears in n malware runs. However, this is also a set of *unique* behavior that *never* appears in m benign runs. The underlying observation from this formula is that kernel malware will consistently perform malicious operations during attacks so we estimate malware behavior by taking the intersection of malicious runs. Such behavior should not occur in benign runs. Therefore, we subtract the union of benign runs from the derived malware behavior.

False positives may occur if a part of a signature is observed in a new tested benign run. The cause of this problem is not unknown kernel behavior, but rather a part of a signature not being properly pruned out in the signature generation. By exercising a variety of workloads in multiple kernel execution instances, we expect that such potential behavior for this error can be significantly reduced due to such constraints.

Generalizing Malware Code Identity. DataGene aims at a solution that can handle rootkits regardless of their attack vectors and can match the variants of rootkits whose signatures are available. For example, DataGene can be used to inspect suspicious data activity in the execution of new signed drivers (which may include hidden malicious code), the execution of an unknown driver (which may be malware or its variant), or kernel execution (where legitimate kernel code can be exploited indirectly for attacks).

In order to cover variants of malicious code, DataGene does not use specific identification of kernel drivers. When we generate or test signatures, we generalize the information specific to kernel drivers, thus allowing signatures to be tested against any driver from new signed drivers to new driver-based rootkits. Specifically, when the signature for a driver-based rootkit is generated, all code sites in this malicious driver are substituted by a single anonymous code site, ε. Some rootkits allocate memory and place their code on it, and any code site in such memory is also generalized as ε. In this process, we also generalize all benign kernel modules in the same way and subtract their memory access patterns from the candidates for the signature to collect only the behavior unique to the malware.

We preserve the code sites in the kernel text. The malware exploiting the legitimate kernel code (e.g., the rootkits using memory devices or return-oriented rootkits) is handled by unique access patterns of legitimate code that are not observed in benign runs. In addition, when we match a malware signature with the data behavior profile of a kernel run, we generalize the driver code in the tested run similarly for comparison.

Matching a Malware Signature with a Kernel Run. The likelihood that a malware program M is present in a tested run r is determined by deriving a set of data behavior elements in S_M which belong to the data behavior profile, D_r. This set I corresponds to the intersection of S_M and D_r[2] (i.e., $I = \{i | i \in S_M \wedge i \in D_r\}$); however, this set may not be symmetric for S_M and D_r because we allow two representations (i.e., an offset and a range of offsets) for the f field of a data behavior element. Algorithm 1 presents how this set I is generated.

Specifically, a data behavior signature S_M and a data behavior profile D_r consist of data behavior elements for all of the static and dynamic data structures. The CheckSignature function in Algorithm 1 compares each element of

[2]The data behavior signature (S_M) is a data behavior profile (i.e., a set of data behavior elements) because it is derived by the intersection and union of data behavior profiles.

Algorithm 1 Algorithm to derive a set of data behavior elements in S_M that belong to D_r.

```
 1: function CheckSignature($S_M, D_r$)
 2:     $I \leftarrow \emptyset$
 3:     for each $e$ in $S_M$ do
 4:         for each $e'$ in $D_r$ do
 5:             if CompareElements($e$, $e'$)= 1 then
 6:                 $I \leftarrow I \cup \{e\}$
 7:             end if
 8:         end for
 9:     end for
10:     return $I$
11: end function
12: function CompareElements($e, e'$)
13:     if $e.c \neq e'.c \vee e.o \neq e'.o \vee e.m \neq e'.m \vee e.i \neq e'i$ then
14:         return 0
15:     end if
16:     if $e.f$ is an offset then
17:         if $e'.f$ is an offset then
18:             if $e.f = e'.f$ then
19:                 return 1
20:             end if
21:         else                                    ▷ $e'.f$ is a range of offsets.
22:             if $e.f \in e'.f$ then
23:                 return 1
24:             end if
25:         end if
26:     else                                        ▷ $e.f$ is a range of offsets.
27:         if $e'.f$ is a range of offsets then
28:             if $e.f \subset e'.f$ then
29:                 return 1
30:             end if
31:         end if
32:     end if
33:     return 0
34: end function
```

S_M and D_r, and returns the set of common elements, I. Two for-loops at lines 3 and 4 generate a pair of elements each from S_M and D_r, and those elements are compared by calling the CompareElements function at line 5.

To consider the two compared elements e and e' as identical, their c, o, m, and i fields first should be equal. Next, their offset fields ($e.f$ and $e'.f$) are compared. Because the offset field can be either of an offset or a range of offsets, there are several cases shown in lines 16-33. If $e.f$ is an offset, it can match either an offset or a range of offsets. If both $e.f$ and $e'.f$ are an offset, their values should be identical. If $e.f$ is an offset and $e'.f$ is a range, they can match if $e.f$ belongs to $e'.f$'s range. If $e.f$ is a range of offsets, it can only match a range of offsets that includes $e.f$.

3. IMPLEMENTATION

DataGene generates the patterns of kernel memory accesses transparently without making changes in the source code of the OS. To implement this feature, we employ virtualization techniques. We used the QEMU [4] virtualizer with the KQEMU optimizer for our implementation. The host machine has 3.2Ghz Pentium D CPU and 2GB RAM. The guest machine is configured with 256MB RAM and the Redhat 8 operating system. This experimental platform is chosen for the convenience of implementation. However, our mechanism is generic and applicable to other operating systems and virtual machine platforms.

We implement the kernel memory mapper and the data aggregator in the VMM. The kernel memory mapper tracks kernel memory allocation and deallocation calls and captures dynamic kernel objects at runtime similar to [21]. When there is a request to the VMM, a data behavior profile can be dumped into a file anytime during the execution of the guest OS. For the purpose of generating a signature, dumping the profile once the OS is completely shutdown is preferred to capture most data behavior. However, to detect kernel malware, the data behavior profile can be generated and periodically compared with the signature while the OS is running.

In our experiments, we measured the quality of signatures whether they trigger false positives as we increased the number of benign runs and malicious runs used for generating malware signatures. We found with five or more sets of benign runs and malicious runs, we could generate the signatures that do not cause false positives in our testings with newly generated benign runs. Therefore, we present the data of these five sets of runs. However, we believe that a more number of runs will certainly improve the quality of signatures and it also depends on dynamic workload performed in each run. In the benign runs, we performed various workload from daily commands to non-trivial application benchmarks. The tested workload includes kernel compilation, `ssh`, `scp`, `lsmod`, `ps`, `top`, `find`, and `ls`. Some workloads were executed for several hours to allow any background administrative operation to be performed. We also used the workload of benign module loading and simple operations of the `/dev/kmem` device (e.g., open and close without overwriting kernel memory).

Among the memory accesses for kernel modules, we exclude the accesses to a kernel module by the same module which correspond to the accesses to a module's local variables. This information is not used to generalize the internal module activity. However, the accesses across modules are used after generalizing the accessing code information. In addition, the kernel data structure `module` having the administrative information regarding a kernel module is mapped to the head of each module's memory. We treat this part of memory as a separate data structure from the remaining module code or data.

4. EVALUATION

In this section we evaluate the effectiveness of our data behavior signatures. First, we extract the signatures of three classic rootkits and match them with benign and malicious kernel runs. Second, we compare the signatures of all of the tested kernel rootkits to determine common data behavior across different rootkits and how such common behavior can be effective in detecting the variants of rootkits. Third, we list specific data elements that are shared by rootkit signatures, which provide an in-depth understanding of the attack operations that are common across kernel rootkits.

4.1 Detecting Kernel Rootkits using Data Behavior Signatures

When a data behavior signature is generated, the information specific to the malicious code is generalized in large. Therefore, we hypothesize that data behavior signatures may be used not only to detect the malware whose signature is available, but also to determine the presence of related malware. In order to validate this hypothesis, we generated the signatures of three representative rootkits, and tested benign kernel runs and malicious kernel runs with 16 rootkits.

To generate malware signatures, we chose three rootkits: `adore 0.38`, `SucKIT`, and `modhide`. The `adore` rootkit has been studied in several rootkit defense approaches [17, 18, 23, 22]. This rootkit has several versions with differences in

Table 1: Properties of benign runs for generating rootkit signatures and testing false positives of signatures.

Properties of a data behavior profile	Kernel run for generating signatures					Kernel run for testing false positives				
	1	2	3	4	5	1	2	3	4	5
# of classes for the dynamic objects	206	204	223	207	223	223	223	223	223	223
# of read code sites for the dynamic objects	10976	10857	12576	11365	12934	12610	12636	12635	13087	13118
# of write code sites for the dynamic objects	4342	4301	4885	4503	5176	4890	4911	4925	5285	5281
# of classes for the static objects	15800	15800	15800	15800	15800	15800	15800	15800	15800	15800
# of read code sites for the static objects	29609	29556	30151	29749	31353	30151	30172	30156	31053	33776
# of write code sites for the static objects	4605	4617	4707	4632	6837	4707	4714	4710	6968	8025
# of false positives	-	-	-	-	-	0	0	0	0	0

Table 2: The number of matched data behavior elements between three rootkit signatures and the kernel runs with 16 kernel rootkits (average of 5 runs, Ad1: adore 0.38, Ad2: adore 0.53, Ad3: adore-ng 1.56, SK: SucKIT).

Signature (S_M)		The number of matched data behavior elements between S_M and the kernel runs with the rootkits shown below ($\|I\|$).															
M	$\|S_M\|$	Ad1	Ad2	Ad3	fuuld	hide_lkm	SK	superkit	hp	kbdv3	knark	linuxfu	Rial	cleaner	kis	modhide	modhide1
Ad1	45	*45*	37	16	0	0	10	10	3	5	20	5	7	0	17	0	0
SK	14797	3	2	1	16	13	*14797*	14767	0	1	2	1	1	0	18	0	1
modhide	3	0	0	0	0	0	0	0	0	0	0	0	0	2	1	*3*	2
Detected		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

features and we chose an old version, 0.38, for the signature to evaluate its effectiveness toward newer rootkit versions (0.53 and 1.56). `SucKIT` is known for its attack vector, the `/dev/kmem` device, that avoids the conventional driver-based mechanism [20]. Several other rootkits followed this trend, using this device while having different goals. `modhide` is a rootkit packaged with the `adore` rootkits to hide them from the list of kernel modules.

We used five kernel runs with rootkits and five benign runs to generate the signatures of such rootkits. Three data behavior signatures of the `adore`, `SucKIT`, and `modhide` rootkits have 45, 14797, and 3 data behavior elements, respectively. `SucKIT` has a significantly high number of elements because it scans kernel memory to collect information about the attack targets (e.g., the system-call table), and this behavior is observed as reading numerous static objects with a variety of offsets. Since we allow an offset or a range of offsets for the offset field (f), the number of data behavior elements can vary depending on the threshold to aggregate the elements. This number is the trade-off between the details of malware behavior and efficiency in the size of malware behavior. We used 15 for this threshold after trying several different numbers. The `modhide` rootkit simply manipulates the kernel module list; thus, it has a few elements.

Using the generated three signatures, we inspected a total of 85 kernel runs: five benign runs, and 80 malicious kernel runs with 16 kernel rootkits (each rootkit was used to generate five malicious runs). These 16 rootkits included the three rootkits used for generating signatures, two newer versions of `adore` rootkits, and the other 11 rootkits. The number of matched elements between the compared signatures and the tested runs are presented in Table 1 (benign runs) and Table 2 (malicious runs), which will be explained below in detail.

Reliability in Benign Runs. Table 1 shows the information about two sets of benign kernel execution instances: five benign runs used for signatures (columns 2-6) and another five benign runs for testing potential false positives (columns 7-11). The top three rows (below the column heading) show information about the dynamic objects, such as the number of classes for the dynamic kernel objects, the number of code sites that read the dynamic kernel objects, and the number of code sites overwriting the dynamic kernel objects. The next three rows have the similar information for the static kernel objects. Since static objects (kernel functions and static data structures) are known at compile time, the "# of classes for the static objects" has the same value in different runs.

The far right-hand five columns show the statistics for the benign runs used for testing false positives of the signatures. In these kernel runs, we generated an additional variety in the workload (e.g., more applications and a heavier load with multiple applications) so that such kernel runs contain more code paths and data operations beyond the kernel runs used for generating signatures. This additional runtime variation results in more code sites for memory accesses (i.e., higher numbers in # of read code sites and # of write code sites).

In this experiment, no false positive cases were found, which confirms that our signature generation procedure captures a reasonably close set of unique data behavior of kernel rootkits and that the tested runs did not contain any data behavior that appears in the signatures.

Detecting Rootkits using Data Behavior Signatures. Malicious kernel runs were next tested by using three signatures to determine any running malware based on the similarity of the data access patterns between the compared signature and the kernel run. We tested a total of 80 kernel runs of 16 rootkits having a variety of targets and attack vectors. For instance, seven rootkits (`fuuld`, `hide_lkm`, `hp`, `linuxfu`, `cleaner`, `modhide`, and `modhide1`) directly manipulate kernel objects (DKOM [6]). Four rootkits (`fuuld`, `hide_lkm`, `SucKIT`, and `superkit`) manipulate kernel memory by using the `/dev/kmem` memory device, among which two rootkits (`fuuld` and `hide_lkm`) directly manipulate only kernel data and do not violate kernel code integrity. Therefore, they are not detected by code integrity-based defense systems [23, 24].

Table 2 presents the number of matched data behavior elements between signatures and kernel runs with rootkits (I). Two left-hand columns show the information about signatures: the name (M) of the rootkit used for the signature and the size of the signature ($|S_M|$). The remaining 16 columns present the number of data behavior elements common in the compared signature (based on the rootkit in the row heading) and the kernel run (where the rootkit in the column heading is active). The presented numbers are the averages of five kernel runs. However, the numbers are consistent in the runs with the same rootkit.

Table 3: The number of common data behavior elements in the combination of rootkit signatures (Ad1: adore 0.38, Ad2: adore 0.53, Ad3: adore-ng 1.56, SK: SucKIT).

M	$\lvert S_M \rvert$	Ad1	Ad2	Ad3	fuuld	hide_lkm	SK	superkit	hp	kbdv3	knark	linuxfu	Rial	cleaner	kis	modhide	modhide1
Ad1	45	*45*	37	16	0	0	10	10	3	5	20	5	7	0	17	0	0
Ad2	53	37	*53*	26	0	0	10	10	3	5	19	4	7	0	19	0	0
Ad3	106	16	26	*106*	0	0	1	1	2	4	9	8	0	3	6	0	0
fuuld	37	0	0	0	*37*	13	16	16	0	0	0	0	0	0	0	0	0
hide_lkm	4827	0	0	0	13	*4827*	13	13	0	0	0	0	0	0	0	0	0
SK	14797	3	2	1	16	13	*14797*	14766	0	1	2	1	1	0	18	0	1
superkit	14783	3	2	1	16	13	14769	*14783*	0	1	2	0	1	0	3	0	2
hp	27	3	3	2	0	0	0	0	*27*	0	1	5	0	0	1	0	0
kbdv3	16	5	5	4	0	0	2	2	0	*16*	4	0	6	0	3	0	0
knark	73	20	19	9	0	0	10	10	1	4	*73*	1	4	0	19	0	0
linuxfu	46	5	4	8	0	0	1	0	14	0	1	*46*	0	0	1	0	0
Rial	57	6	6	0	0	0	5	5	0	2	4	0	*57*	0	10	0	2
cleaner	5	0	0	3	0	0	0	0	0	0	0	0	0	*5*	1	2	2
kis	32097	3	3	4	0	0	17	2	1	1	3	1	2	1	*32097*	1	5
modhide	3	0	0	0	0	0	0	0	0	0	0	0	0	2	1	*3*	2
modhide1	8	0	0	0	0	0	1	2	0	0	0	0	2	2	5	2	*8*
# of effective S_M		10	10	10	3	3	12	11	6	8	10	7	8	4	13	3	6
Max $\lvert$effective $S_M\rvert$		37	37	26	16	13	14769	14766	14	5	20	8	7	3	19	2	5

If the rootkit used for the signature and the rootkit in the tested run are identical, the entire signature was matched giving $|I| = |S_M|$. For example, the signatures of `adore 0.38`, `SucKIT`, and `modhide` rootkits fully match the kernel runs with those rootkits (shown in *italics*). We consider that a tested run includes a potential malware running if one or more signatures have a matched element with the kernel run. In our experiments, all kernel runs with rootkits shared elements with one or more signatures, leading to the detection of 16 kernel rootkits.

4.2 Similarities among Data Behavior Signatures

In the previous section we demonstrated that a variety of rootkits can be detected by using the signatures of a few classic rootkits because they have common data access patterns. In this section we quantitatively measure the similarities in data behavior across rootkits by generating and comparing the signatures of the tested rootkits.

We first generated the signatures of 16 kernel rootkits by applying the set operations (Section 2.3) on five kernel runs with rootkits and five benign kernel runs. Then we calculated the similarities among signatures by applying Algorithm 1 on the combinations of 16 rootkit signatures. Table 3 lists the number of common data behavior elements in such combinations. For a pair of rootkits M_1 in the row heading and M_2 in the column heading, the cross section of the corresponding row and column shows the number of data behavior elements common in two signatures of M_1 and M_2. This number may not be symmetric for M_1 and M_2 because a data behavior element can have two representations for its f field (an offset or a range of offsets). If M_1 and M_2 are the same rootkit, the number of elements is shown in *italics.*

For the rootkit M_2 in the column heading, if positive numbers are listed in the column, the signatures of the rootkits (in the row headings) can be used to determine M_2. The number of such signatures (except S_{M_2} itself) is presented at the second bottom row (# of effective S_M). The maximum size of such signatures is shown in the bottom row (Max |effective S_M|). In our experiments, a rootkit shares its data behavior with 3∼13 of other rootkits (more than seven rootkits in average). The rootkits show similar data behavior not only among close variants (e.g., different versions of `adore` rootkits) but also across the rootkits having different attack mechanisms (e.g., `SucKIT` shows similarities with driver-based rootkits such as `knark` or `kis`).

The similarities of data behavior across rootkits are visualized in Figure 4. A node represents a rootkit signature and an arrow shows the similarity between two signatures using three different arrow types. An arrow from a node M_1 to a node M_2 means that the signature M_1 can be used to determine the rootkit of the signature M_2. This figure illustrates that several groups of rootkits have strong similarities. The family of `adore` rootkits (i.e., `adore 0.38`, `adore 0.53`, and `adore-ng 1.56`) are strongly related in general. The `adore-ng 1.56` is connected to other versions with less strong connections, thick dashed arrows, because in newer `adore` versions (bigger than 1.0 whose name is changed to `adore-ng`), the internal attack vector is substantially changed to use dynamic objects instead of static objects. A group of rootkits using the `/dev/kmem` memory device (i.e., `SucKIT`, `hide_lkm`, `fuuld`, and `superkit`) have a strong relationship to one another. The `SucKIT` and the `superkit` are especially connected by using thick solid arrows because they share a majority of data behavior. Some rootkits have relationships with different kinds of rootkits. For example, the `kis` rootkit is connected to driver-based rootkits such as the `adore` rootkits and the `knark` rootkit; but, it is also closely related to `/dev/kmem` based rootkits such as the `SucKIT`.

As seen in Figure 4, the data behavior is not only common in the family of rootkits or similar kinds, but also is available across different kinds of rootkits. The signatures of these related rootkits can be interchangeably used to detect one another.

4.3 Extracting Common Data Behavior Elements

In this section we demonstrate the details of common rootkit attacks which are systematically extracted based on the similarities in rootkits' data behaviors. The data behavior elements from the signatures of all experimented rootkits are ranked with the order of the appearance in rootkits' signatures (N). The top elements are presented in Table 4 after being classified into several categories.

The first five columns present the contents of data behavior elements (quintuple): the accessing code (c); the kind of memory access (o) such as a read (R: $o = 0$) or a write (W: $o = 1$); the kind of accessed memory (m) such as a dynamic object (D: $m = 0$) or a static object (S: $m = 1$); the accessed memory's class (i), which is converted to a data

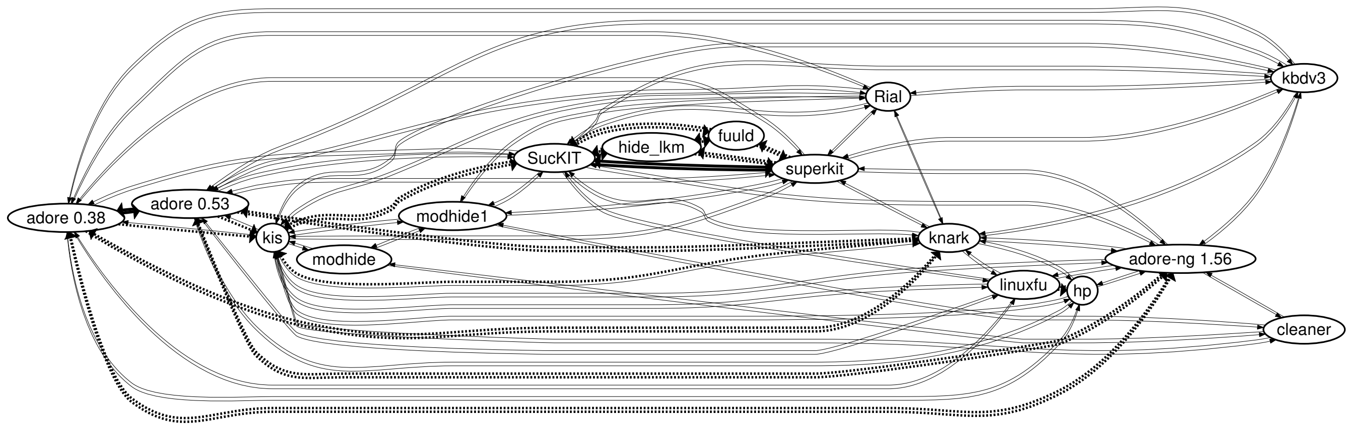

Figure 4: Similarities among the data behavior of rootkits. Types of arrows ($|I|$: # of the matched elements): thin solid ($0 < |I| <= 10$), thick dashed ($10 < |I| <= 30$), and thick solid ($|I| > 30$).

Table 4: Top common data behavior elements among the signatures of 16 rootkits (Ad1: adore 0.38, Ad2: adore 0.53, Ad3: adore-ng 1.56, SK: SucKIT).

Accessing code (c)	o	m	Accessed data (i)	Field,Offset (f)	N	Rootkits with this behavior	Malware behavior
ε	R	D	task_struct	pid	7	Ad1, Ad2, Ad3, hp, knark, linuxfu, kis	Reading a process's ID
ε	R	D	task_struct	flags	6	Ad1, Ad2, Ad3, SK, superkit, knark	Reading a process's flag
ε	W	D	task_struct	uid, euid, gid, egid	5	Ad1, Ad2, Ad3, kbdv3, knark	Privilege escalation
ε	R	D	task_struct	next_task	5	Ad1, Ad2, Ad3, hp, linuxfu	Listing processes
ε	W	D	task_struct	addr_limit	4	Ad1, SK, superkit, kis	Setting an address space information
ε	W	D	task_struct	suid, fsuid, fsgid	4	Ad1, Ad2, Ad3, knark	Privilege escalation
ε	W	D	task_struct	cap_effective	3	Ad1, Ad2, Ad3	Privilege escalation
ε	W	D	task_struct	cap_inheritable	3	Ad1, Ad2, Ad3	Privilege escalation
ε	W	D	task_struct	cap_permitted	3	Ad1, Ad2, Ad3	Privilege escalation
ε	R	D	task_struct	uid	3	Ad1, Ad2, kbdv3	Reading a user's ID
ε	R	D	task_struct	comm	3	Ad1, Ad3, linuxfu	Reading a process' name
ε	W	D	task_struct	next_task, prev_task	2	hp, linuxfu	Hiding a process
read_kmem, write_kmem	R,W	D	file	f_pos	4	fuuld, hide_lkm, SK, superkit	Manipulation via /dev/kmem
memory_lseek	W	D	file	f_pos	4	fuuld, hide_lkm, SK, superkit	Manipulation via /dev/kmem
do_write_mem	R,W	D	file	f_pos	3	fuuld, SK, superkit	Manipulation via /dev/kmem
ε	R	D	module	next	4	kis, cleaner, modhide, modhide1	Scanning the kernel module list
ε	W	D	module	next	3	cleaner, modhide, modhide1	Hiding a kernel module
ε	W	S	sys_call_table	# 141	4	Ad1, Ad2, knark, Rial	Hijacking a system-call
ε	W	S	sys_call_table	# 2,37,120,220	3	Ad1, Ad2, knark	Hijacking a system-call
ε	W	S	sys_call_table	# 6	3	Ad1, Ad2, Rial	Hijacking a system-call
ε	W	S	sys_call_table	# 5	2	Rial, modhide1	Hijacking a system-call
ε	W	S	sys_call_table	# 3	2	knark, Rial	Hijacking a system-call
ε	W	S	sys_call_table	# 59	2	SK, superkit	Hijacking a system-call
__generic_copy_from_user	W	S	sys_call_table	# 59	2	SK, superkit	Hijacking a system-call
ε	W	S	sys_call_table	# 39	2	Ad1, Ad2	Hijacking a system-call
ε	W	S	proc_root_inode_operations	lookup	2	Ad1, Ad2, Ad3	Hijacking a hook on static memory

type for dynamic data or a variable name for static data; and the accessed offset(s) (f). The offset is converted to a field name if it corresponds to a specific field. If the accessed object is the system-call table, a system-call number (#) is designated by dividing the offset by the size of a pointer. The number N and the names of rootkits whose signatures have this element are listed in the next columns. A short description of the element is provided in the far right-hand column by considering the accessed data, offset, and accessing code.

Attacks on Process Control Blocks (PCBs). The first category at the top of Table 4 lists the data behavior that targets the PCBs (type: `task_struct` in Linux). This is a core data structure that maintains administrative information about processes. Therefore it is a major target of rootkits, which aim to manipulate such information.

Table 4 shows that seven rootkits read the process ID numbers in PCBs during attacks. The flags of the processes are accessed by six rootkits. Several rootkits, such as the family of `adore` rootkits, the `kbdv3` rootkit, and the `knark` rootkit, provide a back-door that permits the root privilege to an ordinary user. The `hp` and `linuxfu` rootkits show an attack pattern that manipulates the pointers connecting PCBs. This behavior can hide PCBs from the view inside the operating system.

Attacks using /dev/kmem. The second category shows the rootkit behavior that manipulates kernel memory by using a memory device (e.g., `/dev/kmem`). This device allows a user program to read and write kernel memory like a file putting the kernel integrity at risk. The kernel runs compromised by `fuuld`, `hide_lkm`, `SucKIT`, and `superkit` rootkits commonly show unique data behavior that the kernel functions related to memory devices access `file` kernel objects.

Attacks on the Kernel Module List. The next category lists rootkit attacks on the kernel module list. The `next` pointer field of `module` objects are read or written by the `kis`, `cleaner`, `modhide`, and `modhide1` rootkits. The `module` objects constitute the list of kernel modules and they are connected by this pointer field. The rootkit attacks that hide a module appear as the direct manipulation of this field.

Attacks on Static Kernel Objects. The last category is the manipulation of static kernel objects. Several rootkits hijack the system-calls by replacing the system-call table entries with the addresses of malicious functions. This behavior is captured by the manipulation of the system-call table by several code sites, depending on the attack vector. In the case of driver-based rootkits, such behavior is captured as access by the generalized rootkit code, ε. The rootkits based on memory devices (e.g., `/dev/kmem`) use legitimate kernel code for manipulation (e.g., `__generic_copy_from_user`).

5. DISCUSSION

DataGene is a signature-based approach that detects known and unknown rootkits based on kernel data access patterns similar to the signatures of previously analyzed rootkits. If a rootkit's attack behavior is not similar to any behavior in existing signatures or it does not involve kernel data accesses, such malware is out of coverage of DataGene since such behavior does not match the DataGene's signature.

Many existing rootkits that share the attack goals often exhibit similar data access patterns because essentially these malicious programs generate a false view by manipulating legitimate kernel data structures relevant to the goals. Our approach can detect rootkits by focusing on the common attack targets described in the malware signatures even though such rootkits have different functionalities.

Obfuscating data access patterns involves comparatively more sophistication than code obfuscation because malware requires to use alternate legal code to access kernel data beyond the diversification of malware's own code patterns. Such attack attempts can be detected by employing the defense approaches against control flow anomaly.

DataGene is mainly designed for kernel malware analysis where a potential attack sample is analyzed to determine whether it is malware based on its data behavior. In such an analysis/classification environment with controlled configurations, it is possible to produce no false alarms as presented in our experiments. However, if this technique is further aimed towards a production environment where a diversity of workload could be generated, false alarms may occur due to the foundation of our technique on dynamic execution.

6. RELATED WORK

DataGene introduces a new approach that generates the signature of kernel malware by using their unique data access patterns. There are several approaches related to DataGene in the area of kernel malware analysis and detection.

Malware Defense based on Code Behavior. There has been a variety of approaches which characterizes malware's behavior by using its control flow (e.g., instruction sequences and system-call sequences) [2, 3, 9, 14, 15], and such approaches face the following challenges.

First, malware can obfuscate its execution to elude the code behavior-based malware analyzers. Several papers describe obfuscating techniques such as dead code insertion, code transformation, and instruction substitution [8, 10, 25, 26]. Second, malware's control flow can vary at runtime and the detection mechanism using malware's code behavior should be able to handle such variations.

Complementing these approaches, DataGene uses the pattern of kernel memory accesses, to characterize malware behavior. Because this approach avoids using the control flow of malware, it can be less susceptible to code obfuscation techniques or variations in the malware's control flow.

Kernel Malware Defense based on Code Integrity. The approach based on code integrity [23, 24] allows only authorized kernel code: the kernel text and the kernel modules on a white list. This approach is effective to prevent kernel rootkits that introduce their own code. However, advanced rootkits operate without explicitly injecting malicious code by using techniques such as kernel memory devices, kernel bugs, or return-oriented programming; and this approach cannot handle such cases. DataGene presents a new angle and detects rootkits based on their unique data behavior. Thus it could be applied to such challenging rootkits.

Kernel Rootkit Profilers. Kernel rootkit profilers [22, 27] provide a variety of aspects of rootkit behavior by analyzing rootkit activities and examining user space impact. The profiling result of these approaches is specific to the analyzed malware. In contrast, DataGene uses the malware's memory access patterns whose code information is generalized. Therefore, it has the potential to detect rootkit variants that are similar in data behavior. Also DataGene explores common characteristics across multiple rootkits.

Signatures based on Data Structures. Laika [11] can determine data structures and classify their unique patterns for malware. This approach is effective for user space malware (e.g., botnet programs), which has its own memory space. However, kernel malware's code and data reside in kernel memory together with legitimate kernel code and data. Also kernel malware targets legitimate kernel data and hijacks kernel hooks in addition to using its own data. Therefore, the data behavior in the kernel space is the mixture of the kernel and the kernel malware.

Several approaches [7, 12, 16] can detect kernel data structures based on data properties such as data values and pointer connections. Based on the discovery of data structures, these approaches can also detect kernel rootkits that hide kernel data structures. While the signatures of such approaches are the properties of the data structures, the signatures of DataGene is the properties of malware. Those are generated by using unique data access patterns of malware.

Inter-relationship between Code and Data. SegSlice [5] is a trapping framework that measures and enforces the relationships between the program's code and data units (called slices) using the x86 segmentation system. These relationships are defined by the programmer using SegSlice API. Kernel data access patterns captured by DataGene reflect the relationships between code and data representing which code is expected to access what types of data. These access patterns are systematically captured from the dynamic execution of an operating system kernel by using virtualization technique.

7. CONCLUSION

We have presented a new approach to characterize the behavior of kernel malware by using the patterns of kernel data accesses unique to the malware. The data behavior signature is constructed after generalizing the malware code information. This abstracted data behavior does not use temporal control flow information; therefore, it can match similar data behavior across rootkits and their variants.

Our experiments show that the signatures of three classic rootkits can effectively detect the kernel runs compromised by 16 kernel rootkits and does not trigger any false positives in benign runs. We observe common data behavior across the kernel rootkits in the comparison of their signatures. In addition, we present the details of common data behavior, which provide an in-depth understanding of popular attack behavior of kernel rootkits.

8. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments. This research was supported, in part, by the US Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099, the US Air Force Research Laboratory (AFRL) under Contract FA8750-09-1-0224, and the US National Science Foundation (NSF) under Grant 0716444. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the AFOSR, AFRL, or NSF.

9. REFERENCES

[1] A. Baliga, V. Ganapathy, and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, December 2008.

[2] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[3] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS'09)*, 2009.

[4] F. Bellard. QEMU: A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] S. Bratus, M. E. Locasto, and B. R. Schulte. SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms. In *the 3rd International Conference on Trust and Trustworthy Computing (TRUST'10)*, 2010.

[6] J. Butler. DKOM (Direct Kernel Object Manipulation). `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`.

[7] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, 2009.

[8] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*.

[9] M. Christodorescu, C. Kruegel, and S. Jha. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*.

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages 1998 (POPL'98)*.

[11] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging For Data Structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[12] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, 2009.

[13] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings for the 18th USENIX Security Symposium (Security'09)*, 2009.

[14] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *18th USENIX Security Symposium (Security'09)*, 2009.

[15] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.

[16] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[17] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings for the 13th USENIX Security Symposium (Security'04)*, August 2004.

[18] N. L. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[19] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th conference on USENIX Security Symposium (Security'06)*, 2006.

[20] Phrack Magazine. Linux on-the-fly kernel patching without LKM. `http://www.phrack.com/issues.html?issue=58&id=7`.

[21] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory. In *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection (RAID'10)*, Ottawa, Canada, September 2010.

[22] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th European Conference on Computer Systems (Eurosys'09)*.

[23] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP'07)*. ACM, 2007.

[25] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2009.

[26] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of Software-Based Survivability Mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[27] C. Xuan, J. A. Copeland, and R. A. Beyah. Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In *Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, 2009.